Boosting the efficiency of RISC-V cores: Fine-grain multi-threading and custom instructions, from concepts to implementation

Riadh Ben Abdelhamid[1]

[1] Postdoctoral researcher at the Novel Computing Technologies group, Heidelberg University, Germany

FPGA Ignite Summer School 2024

Outline

Outline

Lecture outline

What I will teach

- **Issues** facing the efficiency of FPGA softcores.
- **Fine-grain multi-threading** and how it improves the efficiency of FPGA softcores.
- Adding **custom instructions** to **riscv-gnu-toolchain** and how they can improve the efficiency of the processor.
- Adding **support for custom instructions** at the RTL level.

What you will learn

- **Well, That is up to you :)**

RISC-V processors

Benefits of using RISC-V

- Linux of hardware (**Open Source ISA** (Instruction Set Architecture)).
- Rich and **growing ecosystem** and **user base**.
- **Modular ISA** with the possibility of using own **custom instructions**.

RISC-V on FPGAs

- Mapping on FPGAs is tricky.
- Conventional Micro-architectures are under performing.

Possible Solutions

- **Barrel Processor** architecture may yield **high compute density**.
- Context storage can be handled by on-chip memories.
- Simpler Deeper Pipeline $\Rightarrow$ **Higher throughput with less logic**.

Efficiency of RISC-V softcores ?

What can we improve ?

Multiple aspects :

- Speed clock speed, throughput, peak performance, sustained performance.
- Area LUTs, BRAMs, etc. $\Rightarrow$ compute density.
- Power Power-efficiency.

How can we improve ?

- **Architecture** (ISA, memory Architecture, etc.).
- **Micro-architecture** (Efficient ISA implementation, Efficient mapping to target hardware, **deep pipelining**, etc.).
- **Tools** (Efficiently using tools like Vivado, Quartus, etc.)

Outline

Challenges of Deep Pipelining on FPGAs

To achieve a **high F_{MAX}**, a softcore processor requires a **deep pipeline**.
Deep pipelining is a technique used to **improve throughput** by breaking down instruction execution into **smaller stages**.

However deep pipelining faces some challenges on FPGAs :

Branch Prediction Penalty

- **Deep pipelines** suffer from **branch prediction penalties**, where **pipeline stages are flushed when branches are mispredicted**.
- FPGAs have **limited resources** to spend on complex branch prediction structures, making efficient prediction challenging.

Increased Forwarding Logic

Deep pipelines require **extensive forwarding logic** to propagate data between pipeline stages. $\Longrightarrow$ **increases resource utilization** and **limits scalability** on FPGAs.

Branch Prediction Penalty and Impact on CPI

To formulate the penalty cost of a branch misprediction in terms of cycles per instruction (CPI), we need to consider the additional cycles incurred due to the misprediction.

CPI cost

The penalty cost can be expressed as :

$$CPI_{overall} = CPI_{correct} + P(misprediction) \times P_m \quad (1)$$

For example, for the case where $CPI_{correct}$=1, where the probability of misprediction is P(misprediction)=0.2 (20% misprediction rate) and where the misprediction penalty is P_m=5 wasted cycles. The resulting $CPI_{overall}$ would evaluate to :

$$CPI_{overall} = 1 + 0.2 \times 5 = 1 + 1 = 2. \quad (2)$$

This means that, on average, **it takes 2 clock cycles to execute each instruction**, considering both correctly predicted branches and the penalty for mispredictions, **which translates to a 100% loss in performance**.

Important Note

The higher the misprediction rate P(misprediction) and/or the Penalty P_m, the greater the impact on the overall CPI, indicating decreased performance due to branch mispredictions.

Data Forwarding (Register Bypassing)

Read After Write (RAW) Hazard (Data Hazard)

This occurs when an instruction needs to read a register that a previous instruction is writing to, and the read would otherwise happen before the write completes. Without bypassing, the pipeline would need to stall until the write completes, as the needed data would not yet be available.

- **Instruction 1 : ADD x3, x1, x2 ;** $\Longrightarrow$ **x3 = x1 + x2**
- **Instruction 2 : SUB x4, x3, x5 ;** $\Longrightarrow$ **x4 = x3 - x5**

In this example, Instruction 2 needs the result of Instruction 1 for the SUB operation. If the processor waits until Instruction 1 writes the result to the register file before allowing Instruction 2 to proceed, this would create a pipeline stall.

Solution with Register Bypassing

Register bypassing allows the result from Instruction 1 (**which will be available at the end of the execute stage**) to be forwarded directly to the input of the execute stage of Instruction 2, **without waiting for the result to be written back to the register file.**

Why Register Bypassing is good ?

Why Register Bypassing is good

- Addresses **RAW hazards** by reducing or eliminating stalls in the pipeline.
- Solves RAW hazards by providing an immediate path for data from the output of one instruction to the input of the next.
- Bypassing improves performance by allowing subsequent instructions to use the results of earlier instructions as soon as they are computed.
- Helps maintaining high instruction throughput and efficient pipeline utilization.

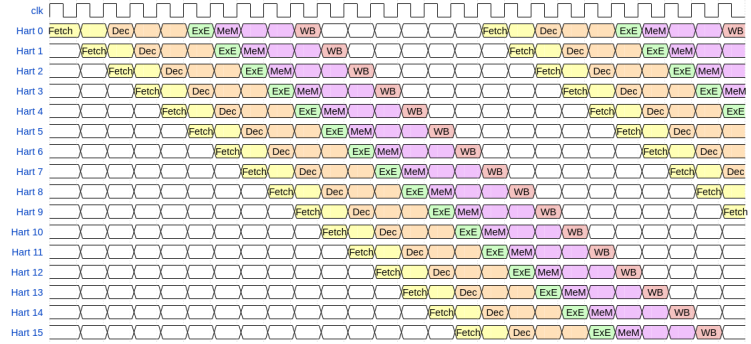
Why Register Bypassing is not good ?

Why Register Bypassing is not good

Data forwarding requires additional hardware in the processor design. Specifically :

- **Multiplexers** : These are used to select the correct data source (either from a register or from an earlier pipeline stage) for each operand of an instruction.
- **Control Logic** : Extra control logic is needed to detect when data forwarding should occur and to control the multiplexers accordingly.

Barrel Processing (Fine-grain multi-threading)



NOTE

- $N_{\text{Hardware Threads}} \geq N_{\text{physical pipeline stages}}$.
- A new Hart is fetched each clock cycle.
- A Hart is executed once every 16 cycles.
- By the time the same Hart is fetched again, all branches and data hazards are resolved $\Longrightarrow$ No need for branch prediction or Register forwarding $\Longrightarrow$ Better MIPS/LUT and higher number of cores is possible

Barrel processor (Fine-Grain Multi-Threading) Advantage

To increase instruction throughput, you must aim for :

- low CPI (≤ 1)
- high maximum clock speed F_{MAX}

Achieving a perfect branch prediction rate nearing 100% contributes to a better CPI.
Additionally, to attain high F_{MAX}, the processor requires a deep pipeline, which typically necessitates register forwarding despite potentially constraining F_{MAX}.

Here, the **barrel processor** comes into play. **Interleaving hardware threads every clock cycle** :

- eliminates the need for branch prediction
- eliminates the need for register forwarding

This effectively allows

- deeper pipeline **without paying** increased branch and forwarding costs.
- Higher clock speed **while maintaining low CPI**

By removing the need for branch prediction and register forwarding, a barrel processor saves logic and results in a more compact implementation $\Longrightarrow$ **Higher compute density (MIPS/LUT)**.

IPS as a performance metric

Instruction Per Second (IPS)

- The CPI metric is **agnostic** to the operating clock speed of a processor.
- The actual processor performance (**Instruction throughput**) can be measured by Equation (3), where **Instruction Per Second (IPS)** is the actual instruction throughput, **Instruction Per Cycle (IPC)** is the inverse of CPI ($IPC = 1/CPI$) and $\mathbf{F_{MAX}}$ is the maximum operating clock speed of the processor.

$$IPS = IPC \times F_{MAX} \tag{3}$$

Important Note

There will be **no need** for register forwarding nor branch prediction, as the pipeline goes deeper, because **when a hart (hardware thread) is re-enabled again**, all data hazards and all branches would be **already resolved**.

Outline

Design of BRISKI (Barrel RISC-V for Kilo-core Implementations)

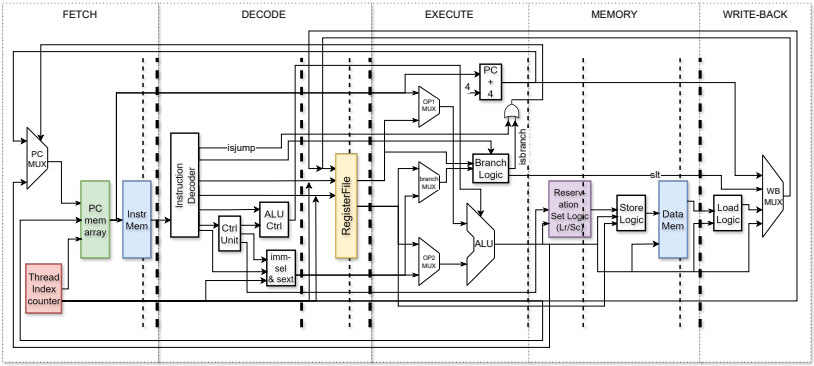


Figure – BRISKI Barrel Processor Architecture.

NOTE

- 16 **RegisterFiles/ProgramCounters** for 16 **Hardware Threads**.
- Fewer than 800 LUTs and fewer than 1K FFs **(near 1-to-1 ratio)**.
- **650+ MHz** (elastic pipeline) $\Longrightarrow$ **650 MIPS** (CPI=1) $\Longrightarrow$ **~0.82 MIPS/LUT**

FPGA Resource Layout and the Need for elasticity pipeline

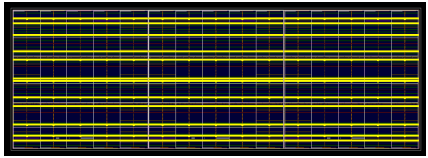


Figure – 12 Columns containing BlockRAM resources (180 BRAMs/Col for a total of 2160 BRAM (2160 RAMB36 or 4320 RAMB18)).

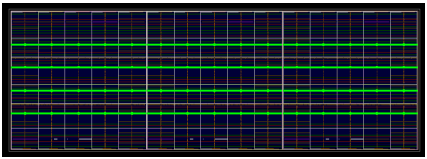


Figure – 4 Columns containing UltraRAM resources (240 URAMs/Col for a total of 960 UltraRAMs).

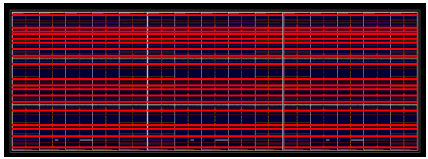


Figure – 19 Columns containing DSP resources (360 DSPs/-Col for a total of 6840 DSPs).

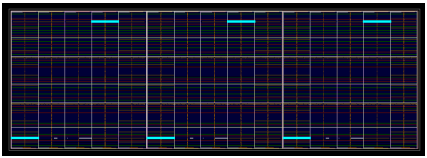


Figure – Columns containing PCIe resources

Design of BRISKI (Barrel RISC-V for Kilo-core Implementations)

NOTE

- One BRAM for **Data / Instructions**.
- One BRAM for **16 register files**.
- Memory Mapped Interface to translate between **load/store** and **control signals**.

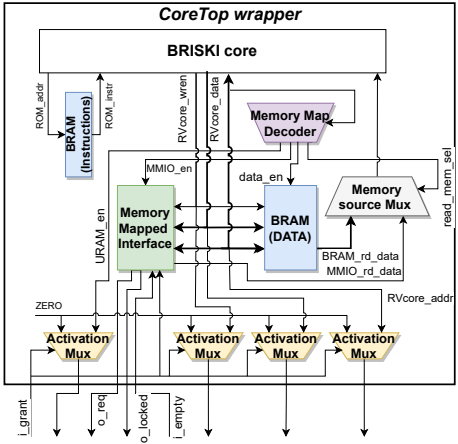


Figure – BRISKI CoreTop Interface wrapper.

Design of BRISKI (Barrel RISC-V for Kilo-core Implementations)

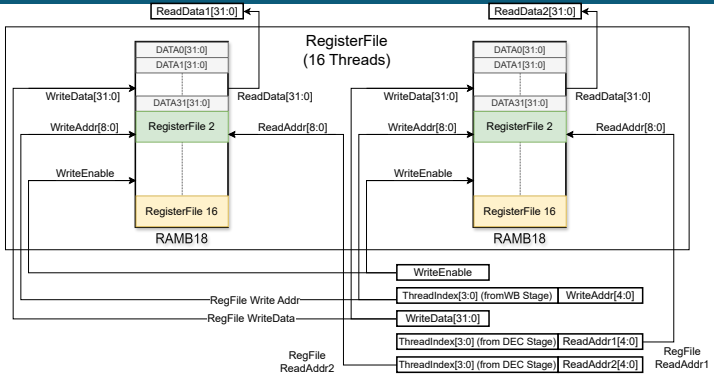


Figure – Register File implementation using two RAMB18 primitives.

NOTE

- **16 register files (2 RAMB18 in SDP mode)**.
- RAMB18 instances with 512 by 32-bit space **fully utilized**.

Compute density

Why Compute density matters

- Compute density : Million of Instruction Per Second Per LUT (MIPS/LUT) ratio
- The short sight goal is to increase MIPS and reduce LUTs $\Longrightarrow$ Improve **Local compute density**.
- The long sight goal is to be able to flood a single FPGA with hundreds of cores without compromising MIPS. $\Longrightarrow$ Improve **Global compute density**.
- The end result would deliver 100s of GIPS on a single FPGA chip.

$$(MIPS/LUT)_{FPGA} = (MIPS/LUT)_{CORE} \times \#CORES \quad (4)$$

In simpler terms, **increasing** Compute density means : More LUTs $\Longrightarrow$ **Increasingly** More MIPS.

State of the art softcore implementations :

TABLE I
COMPARISON WITH RECENT RISC-V RELATED WORKS.

	OoO	bit-serial	in-order	barrel processor
Ref	[1]	[2]	[3]	**BRISKI [4]**
Year	2019	NA	2016	2024
FPGA	XC7Z020	Artix7	VU9P	VU9P
LUT	∼15k	**125**	320	789
FlipFlop (FF)	∼8k	**164**	Not reported	855
BRAM	6	Not reported	0.5-1**	2 (RAMB18)
ISA	RV32IM	RV32I	RV32I+lr/sc-bshift*	RV32I+lr/sc+csrrs
Fmax(MHz)	95.3	220	375	**650**
CPI	NA	>32	1.6	1
MIPS=(Fmax/CPI)	194	∼7	∼234	**650**
Compute Density(MIPS / LUT)	0.012	0.055	0.73	**0.82**

∗ The work [3] reports that Multiply-shift and load/store byte-align sign-extension logic are implemented but shared by core pairs in a cluster.
∗∗ The work [3] reports a BRAM utilization of 4 to 8 in a cluster of 8 cores which leads to 0.5 to 1 BRAM for a single core.

[1] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, K. Inoue, and R. Shioya, “An open source fpga-optimized out-of-order risc-v soft processor,” in 2019 International Conference on Field-Programmable Technology (ICFPT), 2019, pp. 63–71
[2] O. Kindgren. bit-serial risc-v. [Online] https ://github.com/ olofk/serv
[3] J. Gray. (2017) GRVI Phalanx : A Massively Parallel RISC- V FPGA Accelerator Framework A 1680-core, 26 MB Parallel Processor Overlay for Xilinx UltraScale+ VU9P. [Online] : https ://carrv.github.io/2017/papers/gray-phalanx-carrv2017.pdf
[4] https ://github.com/riadhbenabdelhamid/BRISKI

BRISKI enables single-FPGA Kilo Core designs

BRISKI* Barrel Processor Core

- **BRISKI** implements full **RV32I** user mode + atomic extension subset **(LR.W/SC.W)** + **CSRRS**).
- **650+ MHz** on a **VU9P** FPGA.
- **Fewer than 800 LUTs** in most implementations (**Fewer than 700 LUTs** with area optimized directive on a VU9P).
- Current implementation interleaves **16 Hardware Threads**.
- **> 0.8 MIPS/LUT**

[*] **https ://github.com/riadhbenabdelhamid/BRISKI**.

SPARKLE (Scalable Parallel Architecture for RISC-V Kernel-Level Execution)

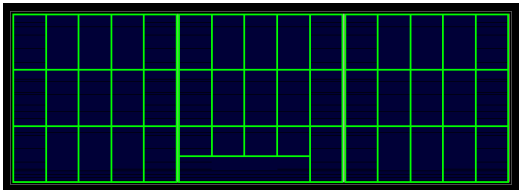


Figure – SPARKLE floorplan on a VU9P FPGA.

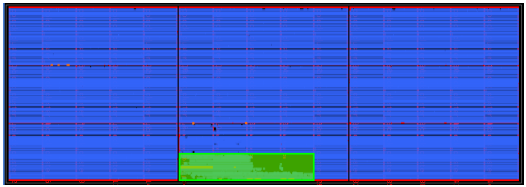


Figure – SPARKLE's Fully placed and routed design, on a VU9P FPGA, with 1,024 BRISKI cores (16,384 Hardware Threads) @400 MHz.

SPARKLE** : 1,024 BRISKI cores @ 400 MHz on a VU9P

- **SPARKLE** is a **scalable** many-core architecture (**scales up and down**).
- Currently running **on a VU9P** with **1,024 BRISKI cores @400MHz and delivering 400 RV32I GIPS**.
- This implementation uses around **800K LUTs**, **2085 BRAMs**, **60 URAMs** and **1,150K FFs**.
- > **0.5 MIPS/LUT**

[**] Riadh Ben Abdelhamid,Vladislav Valek, and Dirk Koch. SPARKLE : A 1024-Core/16,384-Thread single FPGA many-core RISC-V barrel processor Overlay. ASAP 2024.

Outline

PreLab preparation

(Optional) A recommended read

This is a nice guide to add custom instructions, however some of the contents are outdated :
https ://pcotret.gitlab.io/riscv-custom/sw_toolchain.html

Set Up the Tools

- The provided Virtual Machine comes with **verilator** and **riscv-gnu-toolchain** pre-installed.
- If you do not prefer to use or can not use the VM, make sure to have these tools downloaded and installed.

Clone the BRISKI core repo and switch to FPGAIgnite24 branch

- git clone **https ://github.com/riadhbenabdelhamid/BRISKI.git**
- cd BRISKI
- **git switch FPGAIgnite24**

BRISKI Repo structure

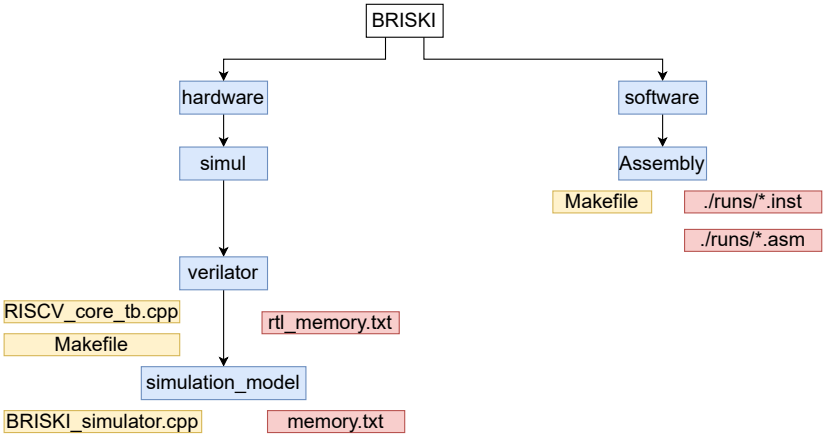


Figure – Structure of the BRISKI github repo.

Convert lower cases to upper cases with fine-grain multi-threading

Open the file **../BRISKI/software/assembly/lower_upper_byte.s**

Listing – Data section

```
1  .section .data
2  .align 4
3  # Data sections for each hart, each containing an array of 32 ASCII characters
4  hart0_data: .ascii "AbcDefGhijKlmNop@#%$&*!()_-+=012"
5  hart1_data: .ascii "zXyWVutsrQpOnMLkjihgfEDCBA987654"
6  hart2_data: .ascii "PqRsTuvWXYzabcdefghij!@#$%^&*()_"
7  hart3_data: .ascii "lmnOpQrStUvWxYzABCDEFGHIJ{}|;:<>"
8  hart4_data: .ascii "KLMNoPQrStUVWXyzO123456789~`-=_+"
9  hart5_data: .ascii "abcdefghijKLMNOPQR2345678901*&^%"
10 hart6_data: .ascii "1234567890abcdefGHIJKLMnoPQRSTuv"
11 hart7_data: .ascii "yzABCDEFghijKLMNOpqrst0123456789"
12 hart8_data: .ascii "ghijKLMNOPQRSTuvwxyZ!@#$%^&*()12"
13 hart9_data: .ascii "ABCDefghijklmnopQRSTuvWXYZ012345"
14 hart10_data: .ascii "mnopQRSTUVWXyzab@#%$&*!()_-+=012"
15 hart11_data: .ascii "xyz1234567890ABCD%$&*(!)_+=-{}|["
16 hart12_data: .ascii "wxyZABCDEfghijklmnopQRSTuvWXYZ12"
17 hart13_data: .ascii "abcdefghijklmNOPQRSTUVWXyz012345"
18 hart14_data: .ascii "PQRSTuvWXYzabcdefghijklmnop!@#%"
19 hart15_data: .ascii "1234567890ABCDXYZefghijklmnopQRS"
20
21 .align 4
22 shared_counter: .word 0 # Shared counter for barrier synchronization
```

Convert lower cases to upper cases with fine-grain multi-threading

Listing – Text section : Initialization

```
1  .section .text
2  .globl _start
3
4  _start:
5      li t0, 32 # Length of the ASCII array (32 characters)
6      la t1, hart0_data # Load address of hart0 data
7      la t2, shared_counter # Load address of shared counter
8
9      # Determine hart id (for simplicity, using a fixed base register)
10     csrr a0, mhartid # Read the hart ID
11     slli a0, a0, 5 # Each hart's data starts 32 bytes apart
12     add t1, t1, a0 # Calculate start of this hart's data section
```

Understanding ASCII characters

CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX
[NUL]	0	00		32	20	@	64	40	`	96	60
[SOH]	1	01	!	33	21	A	65	41	a	97	61
[STX]	2	02	"	34	22	B	66	42	b	98	62
[ETX]	3	03	#	35	23	C	67	43	c	99	63
[EOT]	4	04	$	36	24	D	68	44	d	100	64
[ENQ]	5	05	%	37	25	E	69	45	e	101	65
[ACK]	6	06	&	38	26	F	70	46	f	102	66
[BEL]	7	07	'	39	27	G	71	47	g	103	67
[BS]	8	08	(	40	28	H	72	48	h	104	68
[HT]	9	09	)	41	29	I	73	49	i	105	69
[LF]	10	0A	*	42	2A	J	74	4A	j	106	6A
[VT]	11	0B	+	43	2B	K	75	4B	k	107	6B
[FF]	12	0C	,	44	2C	L	76	4C	l	108	6C
[CR]	13	0D	-	45	2D	M	77	4D	m	109	6D
[SO]	14	0E	.	46	2E	N	78	4E	n	110	6E
[SI]	15	0F	/	47	2F	O	79	4F	o	111	6F
[DLE]	16	10	0	48	30	P	80	50	p	112	70
[DC1]	17	11	1	49	31	Q	81	51	q	113	71
[DC2]	18	12	2	50	32	R	82	52	r	114	72
[DC3]	19	13	3	51	33	S	83	53	s	115	73
[DC4]	20	14	4	52	34	T	84	54	t	116	74
[NAK]	21	15	5	53	35	U	85	55	u	117	75
[SYN]	22	16	6	54	36	V	86	56	v	118	76
[ETB]	23	17	7	55	37	W	87	57	w	119	77
[CAN]	24	18	8	56	38	X	88	58	x	120	78
[EM]	25	19	9	57	39	Y	89	59	y	121	79
[SUB]	26	1A	:	58	3A	Z	90	5A	z	122	7A
[ESC]	27	1B	;	59	3B	[	91	5B	{	123	7B
[FS]	28	1C	<	60	3C	\	92	5C	|	124	7C
[GS]	29	1D	=	61	3D	]	93	5D	}	125	7D
[RS]	30	1E	>	62	3E	^	94	5E	~	126	7E
[US]	31	1F	?	63	3F	_	95	5F	[DEL]	127	7F

CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX
€	128	80		160	A0	À	192	C0	à	224	E0
(n/a)	129	81	¡	161	A1	Á	193	C1	á	225	E1
‚	130	82	¢	162	A2	Â	194	C2	â	226	E2
ƒ	131	83	£	163	A3	Ã	195	C3	ã	227	E3
„	132	84	¤	164	A4	Ä	196	C4	ä	228	E4
…	133	85	¥	165	A5	Å	197	C5	å	229	E5
†	134	86	¦	166	A6	Æ	198	C6	æ	230	E6
‡	135	87	§	167	A7	Ç	199	C7	ç	231	E7
ˆ	136	88	¨	168	A8	È	200	C8	è	232	E8
‰	137	89	©	169	A9	É	201	C9	é	233	E9
Š	138	8A	ª	170	AA	Ê	202	CA	ê	234	EA
‹	139	8B	«	171	AB	Ë	203	CB	ë	235	EB
Œ	140	8C	¬	172	AC	Ì	204	CC	ì	236	EC
(n/a)	141	8D		173	AD	Í	205	CD	í	237	ED
Ž	142	8E	®	174	AE	Î	206	CE	î	238	EE
(n/a)	143	8F	¯	175	AF	Ï	207	CF	ï	239	EF
(n/a)	144	90	°	176	B0	Ð	208	D0	ð	240	F0
'	145	91	±	177	B1	Ñ	209	D1	ñ	241	F1
'	146	92	²	178	B2	Ò	210	D2	ò	242	F2
"	147	93	³	179	B3	Ó	211	D3	ó	243	F3
"	148	94	´	180	B4	Ô	212	D4	ô	244	F4
•	149	95	µ	181	B5	Õ	213	D5	õ	245	F5
–	150	96	¶	182	B6	Ö	214	D6	ö	246	F6
—	151	97	·	183	B7	×	215	D7	÷	247	F7
˜	152	98	¸	184	B8	Ø	216	D8	ø	248	F8
™	153	99	¹	185	B9	Ù	217	D9	ù	249	F9
š	154	9A	º	186	BA	Ú	218	DA	ú	250	FA
›	155	9B	»	187	BB	Û	219	DB	û	251	FB
œ	156	9C	¼	188	BC	Ü	220	DC	ü	252	FC
(n/a)	157	9D	½	189	BD	Ý	221	DD	ý	253	FD
ž	158	9E	¾	190	BE	Þ	222	DE	þ	254	FE
Ÿ	159	9F	¿	191	BF	ß	223	DF	ÿ	255	FF

Figure – Example encoding of ASCII characters.

11. https ://www.ascii-code.net/

Convert lower cases to upper cases with fine-grain multi-threading

Listing – Text section : Convert loop

```
    # Character Conversion Loop
convert_loop:
    lb a1, 0(t1) # Load character from array
    #beqz a1, finish # End of string (null character), exit loop
    li a2, 'a' # Load 'a'
    li a3, 'z' # Load 'z'
    blt a1, a2, next_char # If char < 'a', not a lowercase letter
    bgt a1, a3, next_char # If char > 'z', not a lowercase letter

    # Convert to uppercase
    li a4, 32 # ASCII difference between upper and lower case
    sub a1, a1, a4 # Convert to uppercase
    sb a1, 0(t1) # Store back converted character

next_char:
    addi t1, t1, 1 # Move to next character
    addi t0, t0, -1 # Decrease character count
    bnez t0, convert_loop # Continue loop if more characters
```

Convert lower cases to upper cases with fine-grain multi-threading

Listing – Text section : Barrier and Termination

```
    # Barrier Synchronization
finish:
    li t6, 16 # Total number of harts
    li t3, 1 # Atomic increment value
barrier:
    lr.w t4, 0(t2) # Load current counter value
    add t4, t4, t3 # Increment counter
    sc.w t5, t4, 0(t2) # Store conditionally
    bnez t5, barrier # Retry if SC failed
exit_barrier:
    lw t4, 0(t2) # Total number of harts
    bne t4, t6, exit_barrier # Wait until all harts have reached this point

    # Termination
    ecall # End program (simulated halt for each hart)
```

Using a custom instruction in the convert loop

Open the file **../BRISKI/software/assembly/lower_upper_byte_custom.s**

Listing – Text section : Convert loop with custom instruction

```
    # Character Conversion Loop
convert_loop:
    lb a1, 0(t1) # Load character from array
    #beqz a1, finish # End of string (null character), exit loop
    #li a2, 'a' # Load 'a'
    #li a3, 'z' # Load 'z'
    #blt a1, a2, next_char # If char < 'a', not a lowercase letter
    #bgt a1, a3, next_char # If char > 'z', not a lowercase letter

    # Convert to uppercase
    #li a4, 32 # ASCII difference between upper and lower case
    #sub a1, a1, a4 # Convert to uppercase
    lotoupcase a1, a1, x0 # Custom instruction: a1 = lotoupcase(a1)
    sb a1, 0(t1) # Store back converted character

next_char:
    addi t1, t1, 1 # Move to next character
    addi t0, t0, -1 # Decrease character count
    bnez t0, convert_loop # Continue loop if more characters
```

Linker Description Script (.lds file)

Listing – Defining Memory layout for text and data (lower_upper_byte_custom.lds)

```
1  /* Define memory regions */
2  MEMORY
3  {
4      /* Define RAM and ROM memory regions with specific addresses and sizes */
5      RAM (rwx) : ORIGIN = 0x00000200, LENGTH = 3072
6      ROM (rx) : ORIGIN = 0x00000000, LENGTH = 1024
7  }
8
9  /* Define the sections and their placement */
10 SECTIONS
11 {
12     /* Place the .text section in ROM */
13     .text : {
14         *(.text) /* All .text sections from input files */
15     } >ROM
16
17     /* Place the .data section in RAM */
18     .data : {
19         *(.data) /* All .data sections from input files */
20     } >RAM
21
22     /* Additional sections can be added here */
23 }
```

Makefile commands to generate executable instructions (.inst file)

Custom path of your configured toolchain

- Open **BRISKI/software/Makefile**
- Update USR_BIN to where your custom install path for the riscv-gnu-toolchain (**$(HOME)/summer_school/riscv-custom/newlib/bin**)

Listing – Makefile commands to generate executable instructions (.inst file)

```
1  PROG?=lower_upper_byte
2  RUN_DIR?=runs
3  #USR_BIN?=/usr/bin
4  USR_BIN?=/home/riadh/tools/riscv-newlib-installpath/bin
5
6  hex_gen: clean compile_link objdump_elf
7         python3 hexgen.py $(RUN_DIR)/$(PROG).asm $(RUN_DIR)/$(PROG).inst
8
9  compile_link:
10        mkdir -p $(RUN_DIR)
11        cd $(RUN_DIR) && $(USR_BIN)/riscv64-unknown-elf-gcc -march=rv32iazicsr -mabi=ilp32 -ffreestanding -nostdlib
                -o $(PROG).elf -T ../assembly/$(PROG).lds ../assembly/$(PROG).s
12
13 objdump_elf: compile_link
14        cd $(RUN_DIR) && $(USR_BIN)/riscv64-unknown-elf-objdump -mriscv:rv32 -d -j .text -s -j .data $(PROG).elf > $
                (PROG).asm
```

Cloning and configuring the riscv-gnu-toolchain

Important Note

Skip this if you are using the provided Virtual Machine !

Listing – Pre-requisite packages

```
1 sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk
      build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev device-tree-compiler
```

Listing – Cloning the riscv-gnu-toolchain

```
1 git clone --recurse-submodules https://github.com/riscv/riscv-gnu-toolchain.git
```

Cloning and configuring the riscv-gnu-toolchain

Skip this if you are using the provided Virtual Machine ! (prefix has already been configured to **/home/user/summer_school/riscv-custom/newlib**)

Listing – the toolchain is assumed to be built in /opt/riscv$_{C}$*ustom* :

```
1 cd riscv-gnu-toolchain
2 ./configure --prefix=/home/user/summer_school/riscv-custom/newlib
3 make -j$(nproc)
```

Listing – Check the cross-compiler version

```
1 /home/user/summer_school/riscv-custom/newlib/bin/riscv64-unknown-elf-gcc --version
```

Listing – The riscv-opcodes directory should contain all opcodes

```
1 git clone https://github.com/riscv/riscv-opcodes
```

Understanding RISC-V base instruction formats

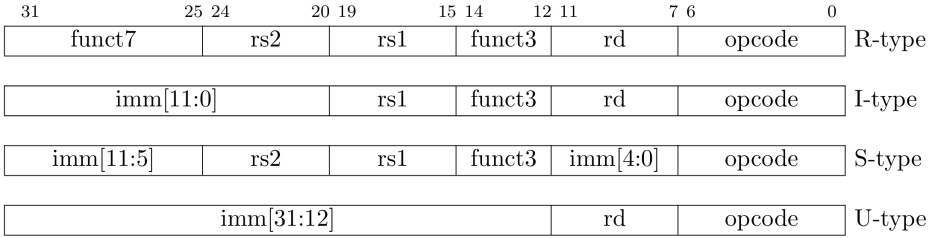


Figure 2.2: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

2

2. https ://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

Understanding RISC-V base instruction formats

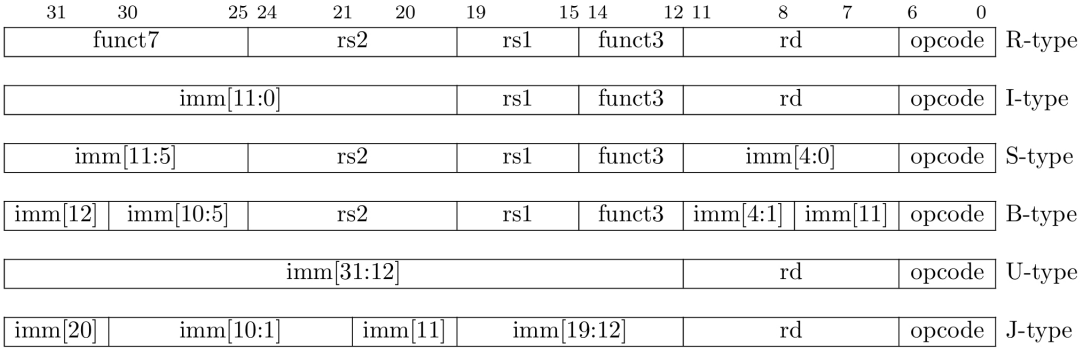


Figure 2.3: RISC-V base instruction formats showing immediate variants.

3

3. https ://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

Understanding RISC-V custom instruction encoding

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Table 24.1: RISC-V base opcode map, inst[1:0]=11

4

4. https ://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

Adding a custom instruction in the cross-compiler

Listing – Some of the opcodes in /home/user/summer_school/riscv-custom/riscv_opcodes/rv_i :

```
# rv_i
lui rd imm20 6..2=0x0D 1..0=3
auipc rd imm20 6..2=0x05 1..0=3
jal rd jimm20 6..2=0x1b 1..0=3
jalr rd rs1 imm12 14..12=0 6..2=0x19 1..0=3
beq bimm12hi rs1 rs2 bimm12lo 14..12=0 6..2=0x18 1..0=3
bne bimm12hi rs1 rs2 bimm12lo 14..12=1 6..2=0x18 1..0=3
blt bimm12hi rs1 rs2 bimm12lo 14..12=4 6..2=0x18 1..0=3
bge bimm12hi rs1 rs2 bimm12lo 14..12=5 6..2=0x18 1..0=3
bltu bimm12hi rs1 rs2 bimm12lo 14..12=6 6..2=0x18 1..0=3
bgeu bimm12hi rs1 rs2 bimm12lo 14..12=7 6..2=0x18 1..0=3
.
add rd rs1 rs2 31..25=0 14..12=0 6..2=0x0C 1..0=3
sub rd rs1 rs2 31..25=32 14..12=0 6..2=0x0C 1..0=3
sll rd rs1 rs2 31..25=0 14..12=1 6..2=0x0C 1..0=3
slt rd rs1 rs2 31..25=0 14..12=2 6..2=0x0C 1..0=3
sltu rd rs1 rs2 31..25=0 14..12=3 6..2=0x0C 1..0=3
```

We will follow the example of add opcode with 3 operands (rd, rs1 and rs2) :

Listing – Adding a custom instruction in /home/user/summer_school/riscv-custom/riscv_opcodes/rv_i :

```
#custom 0
lotoupcase rd rs1 rs2 31..25=1 14..12=0 6..2=2 1..0=3
```

Adding a custom instruction in the cross-compiler

We have to generate MASK and MATCH for the custom instruction

Listing – the opcodes in riscv_opcodes/rv_i :

```
1 make
```

This will generate /home/user/summer_school/riscv-custom/riscv_opcodes/encoding.out.h
Check that file for :

Listing – from /home/user/summer_school/riscv-custom/riscv_opcodes/encoding.out.h :

```
1 #define MATCH_LOTOUPCASE 0x200000b
2 #define MASK_LOTOUPCASE 0xfe00707f
```

Adding a custom instruction in the cross-compiler

MASK

- Bits set to 1 in the MASK indicate positions that are significant and should be matched exactly, while bits set to 0 indicate positions that can vary.
- For example, consider an instruction with a 32-bit encoding. A MASK might look like 0xFFFFF000, which means that the first 20 bits (from the left) of the instruction are significant for the purpose of matching. The last 12 bits can vary without affecting the recognition of the instruction.

MATCH

- When decoding an instruction, the relevant bits (as indicated by the MASK) are extracted from the instruction, and if they match the bits specified by the MATCH value, the instruction is recognized as a specific operation.
- For example, if an instruction's encoding is to be matched against a specific operation, the combination of the MASK and MATCH will be used to identify whether the instruction corresponds to that operation.

How the cross compiler recognizes that an instruction is matched ?

When an instruction is encountered, its relevant bits (as filtered by the MASK) are compared with the MATCH value. If (instruction & MASK) == MATCH, then the instruction is recognized as the specific custom instruction.

Adding a custom instruction in the cross-compiler

Let's Modify the binutils files : **/home/user/summer_school/riscv-custom/riscv-gnu-toolchain/binutils/include/opcode/riscv-opc.h** should be updated to add : (The + sign is indicating added lines and should not be added in your file)

Listing – adding the instruction to the riscv-opc.h

```
1  /* Instruction opcode macros. */
2  + #define MATCH_LOTOUPCASE 0x200000b
3  + #define MASK_LOTOUPCASE 0xfe00707f
4  #define MATCH_SLLI_RV32 0x1013
5  // [...]
6  #endif /* RISCV_ENCODING_H */
7  #ifdef DECLARE_INSN
8  + DECLARE_INSN(lotoupcase, MATCH_LOTOUPCASE, MASK_LOTOUPCASE)
```

Adding a custom instruction in the cross-compiler

The related C source file (**/home/user/summer_school/riscv-custom/riscv-gnu-toolchain/binutils/opcodes/riscv-opc.c**) needs to be updated too : (The + sign is indicating added lines and should not be added in your file)

Listing – adding the instruction to the riscv-opc.c (under riscv-opcodes struct)

```
1 /* name, xlen, isa, operands, match, mask, match_func, pinfo. */
2 + {"lotoupcase", 0, INSN_CLASS_I, "d,s,t", MATCH_LOTOUPCASE, MASK_LOTOUPCASE, match_opcode, 0 },
```

Implementing the custom instruction in the cross-compiler

Listing – rerun make

```
1
2 cd /home/user/summer_school/riscv-custom/riscv-gnu-toolchain
3 make clean
4 make -j$(nproc)
```

If you assigned 'nproc=4' processors to your VM, you can set : **make -j 4**

This will take a while, Grab a coffe !

Checking the custom instruction using the updated cross-compiler

Listing – Sample test

```
1  //Use this sample code to test your custom instruction:
2  #include <stdio.h>
3  int main(){
4      int a,b,c;
5      a = 'a';
6      b = 0;
7      asm volatile
8      (
9      "lotoupcase %[z], %[x], %[y]\n\t"
10     : [z] "=r" (c)
11     : [x] "r" (a), [y] "r" (b)
12     );
13     return 0;
14 }
```

Listing – compile using the newly added custom instruction

```
1  /home/user/summer_school/riscv-custom/newlib/bin/riscv64-unknown-elf-gcc prog.c -o prog
2  file prog
```

Congratulations!!! you just compiled a program using your first **custom instruction**!

Hands-on Lab

Add another custom instruction and recompile the toolchain

Replay the previous steps to implement a custom instruction that **performs the opposite computation** : Converting ASCII characters **from Upper to lower case**.

Recompiling the toolchain will take some 30 mins depending on your machines.

Lets launch the recompilation before the coffee break !

Files that you will use/modify

- ../riscv-custom/riscv_opcodes/rv_i
- ../riscv-custom/riscv-opcodes/encoding.out.h
- ../riscv-custom/riscv-gnu-toolchain/binutils/include/opcode/riscv-opc.h
- ../riscv-custom/riscv-gnu-toolchain/binutils/opcodes/riscv-opc.c

Outline

Remember BRISKI ? (Barrel RISC-V for Kilo-core Implementations)

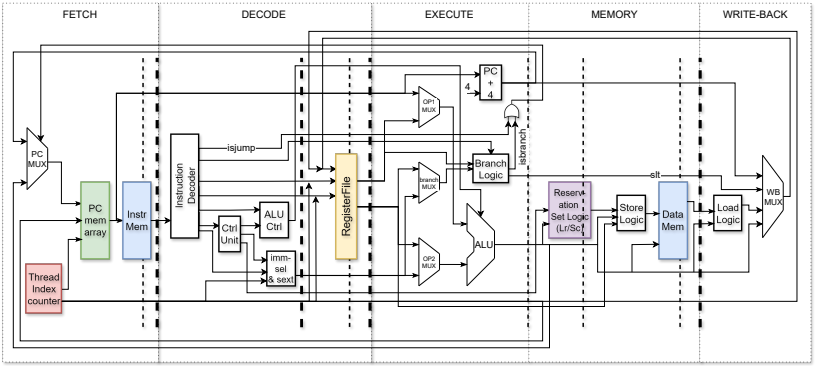


Figure – BRISKI Barrel Processor Architecture.

RTL modules to be updated

- (**control_unit.sv**) and (**alu_control.sv**) and (**alu.sv**)
- **and do not forget riscv-pkg.sv where all parameters reside**.

Modifying riscv-pkg.sv

Listing – riscv_pkg.sv

```
1   //============================
2   // ALU specific params
3   //============================
4   parameter int ALUOP_WIDTH = 4;
5   parameter logic [ALUOP_WIDTH-1:0] ADD_OP = 4'b0000;
6   parameter logic [ALUOP_WIDTH-1:0] SUB_OP = 4'b0001;
7   parameter logic [ALUOP_WIDTH-1:0] OR_OP = 4'b1000;
8   parameter logic [ALUOP_WIDTH-1:0] AND_OP = 4'b1001;
9   parameter logic [ALUOP_WIDTH-1:0] XOR_OP = 4'b0101;
10  parameter logic [ALUOP_WIDTH-1:0] PASS_OP = 4'b1010;
11  parameter logic [ALUOP_WIDTH-1:0] SLT_OP = 4'b0011;
12  parameter logic [ALUOP_WIDTH-1:0] SLTU_OP = 4'b0100;
13  parameter logic [ALUOP_WIDTH-1:0] SLL_OP = 4'b0010;
14  parameter logic [ALUOP_WIDTH-1:0] SRL_OP = 4'b0110;
15  parameter logic [ALUOP_WIDTH-1:0] SRA_OP = 4'b0111;
16  parameter logic [ALUOP_WIDTH-1:0] LOTOUPC_OP = 4'b1011; //Lower to Upper case OP
```

Modifying control_unit.sv

Listing – control_unit.sv

```
1
2       // custom-0-type instructions
3       7'b0001011: begin
4         o_WBSel = 2'b01; //we need to select the output from the ALU.
5         o_regWE = 1'b1; // We need to enable writes to register file
6         o_ALUctrl = 3'b100; //we need to select the desired custom instruction on the alu_control decoder.
7       end
```

Modifying alu_control.sv

Listing – alu_control.sv

```
1
2       3'b100: //custom-0
3       case (i_funct3)
4         3'b000:
5         case (i_funct7)
6                 7'b0000001: o_ALUOp <= LOTOUPC_OP; // lotoupcase
7                 default: o_ALUOp <= '0;
8         endcase
9         default: o_ALUOp <= '0; // Undefined operation
10      endcase
11      default: o_ALUOp <= '0; // Undefined operation
```

Modifying alu.sv

Listing – alu.sv

```
1   logic [DWIDTH-1:0] o_result_lotoupc;
2
3   always_comb begin
4     shamt = i_op2[4:0];
5   end
6
7   always_comb begin
8     o_result_add = (i_aluop == ADD_OP) ? i_op1 + i_op2 : 0;
9     o_result_sub = (i_aluop == SUB_OP) ? i_op1 - i_op2 : 0;
10    o_result_sll = (i_aluop == SLL_OP) ? i_op1 << shamt : 0;
11    o_result_xor = (i_aluop == XOR_OP) ? i_op1 ^ i_op2 : 0;
12    o_result_or = (i_aluop == OR_OP) ? i_op1 | i_op2 : 0;
13    o_result_and = (i_aluop == AND_OP) ? i_op1 & i_op2 : 0;
14    o_result_pass = (i_aluop == PASS_OP) ? i_op2 : 0;
15    o_result_srl_sra = (i_aluop == SRL_OP || i_aluop == SRA_OP) ? temp : 0;
16    o_result_lotoupc = (i_aluop == LOTOUPC_OP)? (((i_op1 < 97) || (i_op1 > 122))? i_op1 : i_op1-32) : 0;
17  end
18
19  always_ff @(posedge clk) begin
20    o_result <= o_result_add ^ o_result_sub ^ o_result_sll ^ o_result_xor ^ o_result_srl_sra ^ o_result_or ^
          o_result_and ^ o_result_pass ^ o_result_lotoupc;
21  end
```

Outline

Adding the custom instruction to the software simulator for testing

Listing – Where to add a custom instruction in the BRISKI software simulator.

```
1    switch (opcode) {
2          // -- This a custom instruction that reads the contents of register[rs1], check if it is a lower case
                letter then
3          // subtracts 32 to make it upper case
4       case 0x0B: // custom-0 type (opcode = 0b0001011)
5              switch (funct3) {
6                     case 0x0: //(funct3 = 0b000)
7                            if (funct7=0x01){ // (funct7 = 0b0000000) lower to upper case byte
8                               //if ((registers[hart_id][rs1] > 'z') && (registers[hart_id][rs1] <'a')) {// not a
                                      lower case
9                               if ((registers[hart_id][rs1] > 122) || (registers[hart_id][rs1] < 97)) {// not a
                                      lower case
10                              } else {
11                                 registers[hart_id][rd] = registers[hart_id][rs1] - 32;
12                              }
13                           }
14                           break;
15                    default : ; break;
16             }
17          pc[hart_id]+=4;
18          break;
```

Adding the custom instruction to the software simulator for testing

Listing – running the automated check

```
1
2  $ pwd
3  ../[BRISKI]
4  $ cd hardware/simul/verilator/
5  $ make check_all
```

How it is checked ?

- If everything is correctly compiled, the last command should generate a memory dump of the rtl design by using verilator (rtl_memory.txt) and another memory dump of the software simulator memory (./simulation_model/memory.txt) after g++ compilation.
- A simple diff command is called to compare both memory dumps.
- If everything is matching you will get an OK, otherwise, it will display a failing message.
- If it fails, try vimdiff to check which memory addresses differs. This can give you hints to debug. **GOOD LUCK !**
- If it succeeds, you succeeded in adding your first custom instruction. **CONGRATULATIONS !**

Outline

Advanced follow-up Lab

A more challenging example

Take this challenge if :

- Adding a custom instruction was **a piece of cake for you**.
- You crave challenges and enjoy struggles in your life.

Your next task would be to add a more efficient custom instruction.
This instruction allows you to :

- Select either upper-to-lower or lower-to-upper-case. You can use the second register rs2, to specify the desired behavior.
- Convert up to four bytes, in one go. You can use the second register rs2, to specify how many bytes to convert from your provided word aligned address in rs1.
- **Happy Hacking !**

Thank you for your attention !